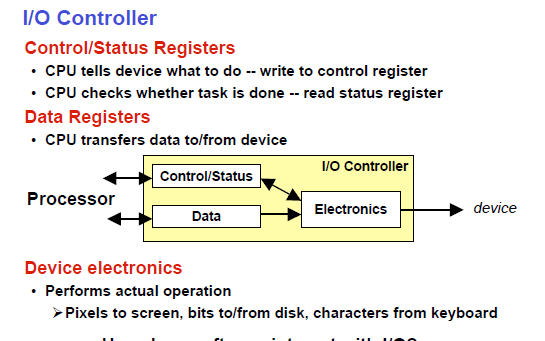
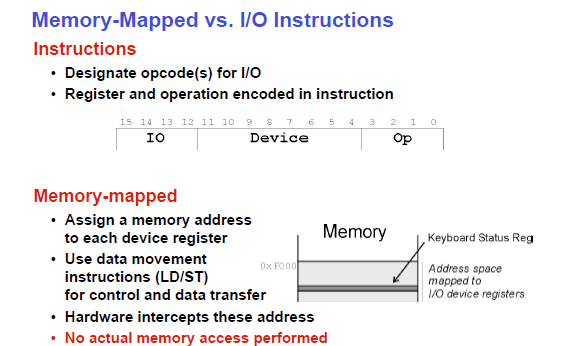
**Interfacing I/O Devices**

****

****

**Giving commands to I/O Devices**

Must write to OR read from device registers, i.e. must communicate commands to the controller. Note that a controller normally contains a microprocessor. When we say the processor below, we mean the central processing unit (not the one on the controller).

* The controller has a few registers that can be read and/or written by the processor, similar to how the processor reads and writes memory. These registers are also read and written by the controller.
* Nearly every controller contains
  + A **data register**, which is readable (by the processor) for an input device (e.g., a simple keyboard), writable for an output device (e.g., a simple printer), and both readable and writable for input/output devices (e.g., disks).
  + A **control register** for giving commands to the device.
  + A readable **status register** for reporting errors and announcing when the device is ready for the next action (e.g., for a keyboard telling when the data register is valid, and for a printer telling when the character to be printed has be successfully retrieved from the data register).
* Many controllers have more registers

**Communicating with the Processor**

Should we check periodically or be told when there is something to do? Better yet can we get someone else to do it since we are not needed for the job?

* We get mail at home once a day.
* At some business offices mail arrives a few times per day.
* No problem checking once an hour for mail.
* If email wasn't buffered, you would have to check several times per minute (second?, milisecond?).
* Checking email this often is too much of a burden and most of the time when you check you find there is none so the check was wasted.

**Polling**

Processor continually checks the device **status register** until new dataarrives or device ready for next data.

* Like the mail example above.
* For a general purpose OS, one needs a timer to tell the processor it is time to check (OS issue).
* For an embedded system (microwave) make the checking part of the main control loop, which is guaranteed to be executed at a minimum frequency (application software issue).
* For a keyboard or mouse, which have very low data rates, the system can afford to have the main CPU check. We do an example just below.
* It is a little better for slave-like output devices such as a simple printer. Then the processor only has to poll after a request has been made until the request has been satisfied.

**Polling Examples**

* Cost of a poll is 400 clocks.
* CPU is 500MHz.
* How much of the CPU is needed to poll
  1. A mouse that requires 30 polls per sec?
  2. A floppy that sends 2 bytes at a time and achieves 50KB/sec?
  3. A hard disk that sends 16 bytes at a time and achieves 4MB/sec?

1. **Mouse Calculation**

A mouse that requires 30 polls per sec:

CPU: 500Mhz = 500,000,000 cycles / second

Poll mouse: 400 clocks for poll.

30 polls X 400 clocks = 12,000 clocks in every second spent polling the mouse.



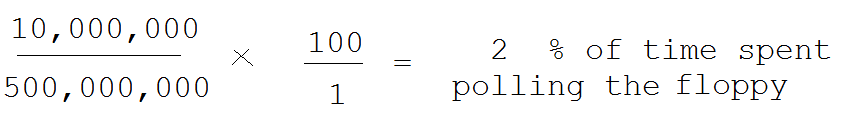
1. **Floppy Calculation**

A floppy that sends 2 bytes at a time and achieves 50KB/sec:

50KB / 2 = 50000(approx) / 2 = 25000 (two byte) packets / second

This means 25,000 polls that will be done every second.

25,000 \* 400 = 10,000,000 cycles



1. **Hard disk Calculation**

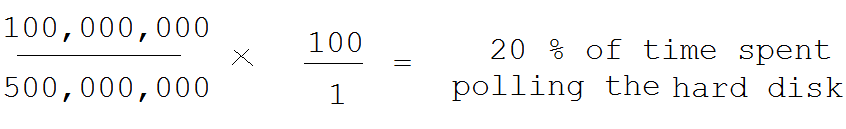
A hard disk that sends 16 bytes at a time and achieves 4MB/sec

4MB / 16 = 4,000,000(approx) / 16 = 250,000 (16 byte) packets /secondThis means 250,000 polls that will be done every second.

(10 Times as many polls as the previous example – so the answer will be 10 times as much i.e. 20%)

Anyway we will work through it…

250,000 polls X 400 cycles for a poll = 100,000,000 cycles



* For the mouse, we use 12,000 clock cycles each second sec for polling. The CPU runs at 500\*10^6 cycles/sec. So polling the mouse requires 12/500\*10^-3 = 2.4\*10^-5 of the CPU. A very small penalty.
* The floppy delivers 25,000 (two byte) data packets per second so we must poll at that rate not to miss one. CPU cycles needed each second is (400)(25,000)=10^7. This represents 10^7 / 500\*10^6 = 2% of the CPU
* To keep up with the disk requires 250K polls/sec or 10^8 clock cycles or 20% of the CPU.
* The system need not poll the floppy and disk until the CPU had issued a request. But then it must keep polling until the request is satisfied.

**Interrupt driven I/O**

Processor is told by the device when to look. The processor is *interrupted* by the device.

* Dedicated lines (i.e. wires) on the bus are assigned for interrupts.
* When a device wants to send an interrupt it asserts the corresponding line.
* The processor checks for interrupts after each instruction. This requires ``zero time'' as it is done in parallel with the instruction execution.
* If an interrupt is pending (i.e., if a line is asserted) the processor.
  1. Saves the PC and perhaps some registers.
  2. Switches to kernel (i.e., privileged) mode.
  3. Jumps to a location specified in the hardware (the *interrupt handler)*.

At this point the OS takes over.

* What if we have several different devices and want to do different things depending on what caused the interrupt?
* Use **vectored** interrupts.
  1. Instead of jumping to a single fixed location, the system defines a set of locations.
  2. The system might have several interrupt lines. If line 1 is asserted, jump to location 100, if line 2 is asserted jump to location 200, etc.
  3. Alternatively, the system could have just one line and have the device send the address to jump to.
* There are other issues with interrupts that are taught in OS. For example, what happens if an interrupt occurs while an interrupt is being processed. For another example, what if one interrupt is more important than another. These are OS issues and are not covered in this course.
* The time for processing an interrupt is typically longer than the type for a poll. But interrupts are *not* generated when the device is idle, a big advantage.

**Interrupt Example**

* Same hard disk and processor as above (i.e. A hard disk that sends 16 bytes at a time and achieves 4MB/sec)
* Cost of servicing an interrrupt is 500 cycles.
* The disk is active only 5% of the time.
* What percent of the processor would be used to service the interrupts?

500 Mhz = 500,000,000 cyles

Interrupt takes 500 clocks

Disk active 5% of time

4MB

--- = 250,000 Interrupts.

16 bytes

(An interrupt will be done on each "data transfer" of 16 bytes)

250,000 x 500 = 125,000,000 cycles

125,000,000 100

----------- x --- = 25% (This figure assumes disk is active all the time)

500,000,000 1

But Disk is active only 5% of the time.

5% of 25 = 1.25

=> 1.25% is the true cost of interrupt handling

* Cycles/sec needed for processing interrupts while the disk is active is 125 million.
* This represents 25% of the processor cycles available.
* But the true cost is only 1.25%, since the disk is active only 5% of the time.
* Note that the disk is not active (i.e., actively generating interrupts) right after the request is made. During the seek and rotational latency, interrupts are not generated. Only during the transfer are interrupts generated.

**Direct Memory Access**

The processor initiates the I/O operation then something else takes care of it and notifies the processor when it is done (or if an error occurs).

* Have a DMA engine (a small processor) on the controller.
* The processor initiates the DMA by writing the command into data registers on the controller (e.g., read sector 5, head 4, cylinder 123 into memory location 34500)
* For commands that are longer than the size of the data register(s), a protocol must be used to transmit the information.
* The controller collects data from the device and then sends it on the bus to the memory without bothering the CPU.
  + So we have a multimaster bus and need some sort of arbitration.
  + Normally the I/O devices are given higher priority than the CPU.
  + Freeing the CPU from this task is good but isn't as wonderful as it seems since the memory is busy (but cache hits can be processed).
  + A big gain is that only one bus transaction is needed per bus load. With PIO, two transactions are needed: controller to processor and then processor to memory.
  + This was for an input operation (the controller writes to memory). A similar situation occurs for output where the controller reads from the memory). Once again one bus transaction per bus load.
* When the controller detects that the I/O is complete or if an error occurs, it sets the status register accordingly and sends an interrupt to the processor to notify the latter that the I/O is complete.

**DMA - Modes of operation**

1) Cycle Steal:

A read or write signal is generated by the DMAC (Direct memory access controller), and the I/O device either generates or latches the data. The DMAC effectively steals cycles from the processor in order to transfer the byte, so single byte transfer is also known as cycle stealing.

2) Burst Transfer:

To achieve block transfers, some DMAC's incorporate an automatic sequencing of the value presented on the address bus. A register is used as a byte count, being decremented for each byte transfer, and upon the byte count reaching zero, the DMAC will release the bus. When the DMAC operates in burst mode, the CPU is halted for the duration of the data transfer.

3) Hidden DMA (Also called **transparent**):

It is possible to perform hidden DMA, which is transparent to the normal operation of the CPU. In other words, the bus is grabbed by the DMAC when the processor is not using it. The DMAC monitors the execution of the processor, and when it recognises the processor executing an instruction which has sufficient empty clock cycles to perform a byte transfer, it waits till the processor is decoding the op code, then grabs the bus during this time. The processor is not slowed down, but continues processing normally. Naturally, the data transfer by the DMAC must be completed before the processor starts

**See Also**

Slide 7 from

http://www.cs.nthu.edu.tw/~ychung/slides/Virtualization/VM-Lecture-2-3-IO%20Virtualization.pptx

[**http://www.electronics.dit.ie/staff/tscarff/DMA/dma.htm**](http://www.electronics.dit.ie/staff/tscarff/DMA/dma.htm)**,**

[**https://en.wikipedia.org/wiki/Direct\_memory\_access**](https://en.wikipedia.org/wiki/Direct_memory_access) **and google books**

Microprocessor 8085 and Its Interfacing p249-p251

**References**

<http://www.cs.nyu.edu/courses/fall00/V22.0436-001/lectures/lecture-26.html>

<https://www.cis.upenn.edu/~milom/cse240-Fall06/lectures/Ch08.pdf>

<http://www.cs.nyu.edu/courses/fall07/V22.0436-001/class-notes.html>

<http://www.electronics.dit.ie/staff/tscarff/DMA/dma.htm>

<https://en.wikipedia.org/wiki/Direct_memory_access>

<http://www.xml.com/ldd/chapter/book/ch08.html>

<https://classes.soe.ucsc.edu/cmpe012/Winter09/lectures/09_LC3_IO.pdf>

-----------------------------------------------------------------------------

Google: kbsr bios

<http://people.cs.georgetown.edu/~squier/Teaching/SystemsFundamentals/121-2012-CourseDocuments/Lec-1c-LC3-OS.pdf> (Interesting)

<http://www.cs.unca.edu/~brock/classes/Spring2002/255/lectures/Lect08.pdf>